PARA PROG

Simone Guggiari
Zusammenfassung 2015

	control HW	power	freedom
single core	complex (bigger)	inefficient	✓ flexibility/performance
multi core	easy	✓	restrictive, difficult programming

parallel
more resources to solve faster
(in reference to specific platform)
WORK ⇄ resources (how to perform / orchestrate)

concurrent
execution of activities at same time, coordination
REQUEST → resource (correctly/efficiently manage resources)

distributed
physical separation

von Neumann Architecture: data & program in same memory

PARALLEL STRATEGIES

vectorization (data parallel model)
do n same operations
at a time: load → OP → store

pipelining (multiple input)
load in | full utilization | load out
5 elem. stages (CPU functional units)
instr. fetch · decode · execute · data access · write back

ILP (instr. level parall)
se le istruzioni non dipendono tra loro (istruz. inutile può bassa) fatte in parallelo.
ADD $ $ $
LW ....

Moore: # trans. double every 18m.
- ⚠ power dissipation
- ⚠ mem. access speed
- can't keep up, add cores

ARCHITECTURES

shared mem: CPU, mem
distributed mem: network

- STT (simultaneous multithreading) 1 core (threads, es. time split)
- multicores (CMP, chip multiprocessing) 1 chip, mult. cors
- SMP (symmetric multiprocessing) mult chips (cache coord)
- NUMA (non uniform mem. access) = distributed

Flynn's taxonomy (architecture classification)

	sequential	~ weird
	a+b=c (CPU)	a(+,-,*,/)b=(c1,c2,c3,c4)
	SISD	MISD
	SIMD	MIMD
	(a0,a1,a2,a3)+(b0,b1,b2,b3)=(c0,c1,c2,c3) (GPU)	(a0,a1,a2,a3)(+,-,*,/)(b0,b1,b2,b3) distrib. syst

vector processing
SMT (multiple threads)
VLIW (very long instr. word)

Formulas performance (time) speedup
(if linear = perfection, no overheads)
serial part = problem / extra computation

$T_p \geq T_1/p$
$S_p \leq \frac{T_1}{T_p}$

Amdahl: (b = sequential)
$T_p = T_1(b + \frac{1-b}{p})$
$S_p \leq \frac{p}{1+b(p-1)} = \frac{1}{b+\frac{1-b}{p}}$

Gustafson (run time const)
$T_1 = p(1-b)T_p + bT_p$
$S_p = p - b(p-1)$

∃ scheduling with (greedy)
$T_p \leq (T_1/p) + (T_\infty)$
$= \frac{T_1 - T_\infty}{p} + T_\infty$
W_{par} absolute, W_{seq} mine

scheduling (assigning work to CPU)

TASK PAR: user expresses tasks (generic, not product)
DATA PAR: " operation, system splits
work partitioning / task decomp: coarse (big) · fine

Graphs (DAG)
sequential (span, critical path)
D&C: recursive splitting
directed acyclic graph
⚠ remember arrows!
graph can be · dynamic (but some tasks still depend on previous) · static (simple data flow)
in → out
parallelizable (wide is better)
split → join

Managing states:
- immutability (data doesn't change) → no problem
- isolated mutability: only its context can access it
- mutable → potential errors (protect state to avoid concurrent access!)

RPM (russian peasant method)

$a \cdot b = \begin{cases} 2a \cdot \frac{1}{2}b \\ a + (2a \cdot \frac{b-1}{2}) \end{cases}$

15	18
*7	36
*3	72
1	144

⇒ 15·18 = 144 + 72 + 36 + 18 = 270
* → remember to add it back

to avoid failure, ensure correctness of code! → USE INVARIANTS

JAVA JVM, interpreted language
avoid redundancy (repetition of code)
control flow (jumps to called method)
↳ also means if/else/loops/...

types

byte	8	float	32
short	16	double	64
int	32	char	16
long	64	bool	1

(INT) (reals)

loops: definite → know how many times will be executed
indefinite → doesn't "
Sentinel value: when seen (from input), does something (fencepost, OBA off by 1 mistake)

Arrays are of type reference!
String: array of char (to compare, use .equals(s), NO ==)

exceptions
try { // protected code }
catch (Exception e1) { // what to do }
finally { // this is always executed (even if quit due to exception) }
inherit from throwable
throw new exception
they happen at runtime
es. · IllegalArgumentException · NullPointerException · IllegalStateException · ArithmeticException ...

Thread is independent source of execution, executes task (each has priority 1-10)
implements Runnable (java.Lang.Runnable)
override run(), put inside what you want to execute
call start() / join()

NEW → RUNNABLE ⇄ running, not runnable
RUNNABLE → terminated

ExecutorService
```
cx = Executors.newFixedThreadPool(n);
ex.execute(task);
ex.shutdown();
```

Fork Join (thread pool, si basa su executor service)
ForkJoinPool
```
fjp = new ForkJoinPool();
fjp.invoke(task)   //@override compute
invokeAll(task1, task2, ...)
```

L.14 - SYNC: beyond locks 1

mem. reordering: processor allows to change order of single op if they (apparently) do not modify program (do not affect semantics of sequential code)
eg. because of pipeline / cache
there is no guarantee of global ordering
⚠ while (x==1); because while (true); because compiler doesn't know other thread can change x

interleaving: program is not executed in seq. order like it was programmed (thread splitted execution)

synchronized: keyword in java used to avoid data races & inconsistency

volatile (used to declare variable, for experts) → ask java to keep variable up to date

atomic op: when doing something that depends on read, make sure they can't be split

DEADLOCK: two or more processes are mutually blocked because each process waits for another of these processes to free a resource

(A) thread
[x] lock (for resource)
(P) → [x] thread P tries to acquire a
[x] → (Q) res. x is held by Q

eg. Bank transfer

lock set	lock self
lock B	lock A

DL occurs when the graph of dependencies between threads-res contains a cycle
DL detection looks for that
DL avoidance can be achieved by · two phase locking / · res. ordering

res. ordering: start acquiring resources with bigger # (or thread w/ bigger id) and set free smallers (so that others can use them)
→ if ordering not available, generate it
You can document order for thread using eg" when doing this, never try to acquire this lock while holding that Lock"
for trees: if holding a node's lock, you may only acquire node's children lock (use DataStruct to define order)

starvation repeated but unsuccessful attempt of recently unblocked process to continue its execution

READERS / WRITERS LOCK

RAR	→ no problem, can allow multiple (simultaneously) (if lookup doesn't modify datastruct)
WAR / RAW	→ problem
WAW	→ problem
USE R/W lock:	(java.util.concurrent.locks. ReentrantReadWriteLock)

states
- not_held
- held_writing
- held_reading

operations
- new()
- acquire_write() → only possible if not held. if held by readers, don't allow new readers to acquire (use starvation)
- release_write()
- acquire_read()
- release_read() → always possible if held by readers. increase readers count by 1

SEMAPHORES
↳ allow thread to communicate about changes
(normal locks are good for mutex & atomicity)
is datastruct, has INT, initial value $S \geq 0$ (how many can happen simultaneously)
- acquire (S): S--
- release (S): S++

$S = 0$ → locked
$S > 0$ → unlocked (S threads can be let into the critical section)

P (wait/acquire)
V (signal/release)
can be used for **rendez-vous**

Semaphores P_arrived, Q_arrived

	P	Q
init	P.a = 0	Q.a = 0
pre:		
sync	P.a.release()	Q.a.release()
	Q.a.acquire()	P.a.acquire()
post		

releasing before acquiring = best solution

Race condition: execution order of code, or time propagation for variable impacts its value
→ BAD, we don't want it!

Barrier: it's like a rendez-vous, but for multiple processes
SOLUTION: 2 phase barrier
init: mutex=1, barA=0, barB=1, count=0

```
atomic {
    acquire(mutex)
    count++;
    if(count==n) acquire(barB); release(barA);
    release(mutex)
}
acquire(barA); release(barA); // barA=1, barB=0 for all processes
atomic {
    acquire(mutex)
    count--;
    if(count==0) acquire(barA); release(barB);
    release(mutex)
}
acquire(barB); release(barB); // barB=1, barA=0 for all processes
```

a barrier for a group of threads: in the source code means only that this point and all other threads must stop at this point and cannot proceed until all other threads reached it

L.15 - SYNC: beyond locks 2

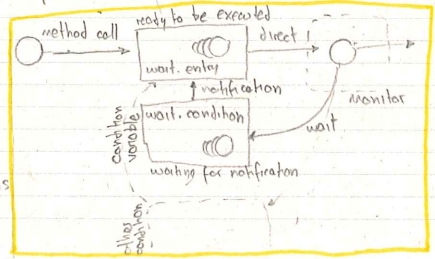
livelock: competing threads are able to detect potential DL but make no progress while trying to solve it (polite lock)

PRODUCER - CONSUMER pattern
To computes x, gives it to T1 that uses it
input = q_in.dequeue();
output = do_smth(input);
q_out.enqueue(output);
semaphores are not sufficient for P/C queues (unstructured, require discipline, because easy to DL)

circular buffer
in, out
one space has to be left empty
empty: in = out

MONITOR
contemporarily escape from lock when waiting on a condition
is abstract datastruct equipped with operations that run in mutex they provide, other than mutex:
check for condition: if doesn't hold → release monitor lock → wait for it to be true
operations: wait() // until signaled
notify() // wakes up 1 thread
notifyAll()
→ signal system (avoid busy looping, waste of clk)

monitor queues:
signal: release mutex, move to wait_queue, sleep thread
queue of threads



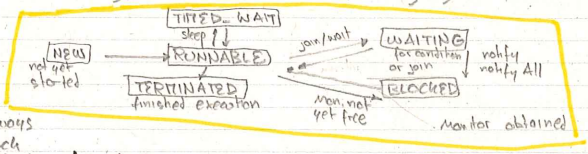
- signal&wait() → exit monitor, go to waiting entry queue
- signal&continue() → continues processing

in JAVA: final Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
(monitor.tryEnter)

await(); signal(); signalAll(); ex Pelse

Sleeping barber variant: use additional counters to know if processes are waiting, so that it can avoid sending signals if not useful

thread states



conditions are always associated with lock

P/C queue with monitor lock

```
Class Queue {
 int in=0, out=0, size;
 long buf[];
 final Lock lock = new ReentrantLock();
 final Condition notFull = lock.newCondition();
 final Condition notEmpty = lock.newCondition();
Queue(int s){   // constructor
  size = s;
  buf = new long[s];
 }
```

```
void Enqueue(long x) {
 lock.lock();
 while (isFull())
  try { notFull.await(),
  } catch (InterExc e){}
 doEnqueue(x);
 notEmpty.signal();
 lock.unlock();
}
```

```
long dequeue() {
 long x;
 lock.lock();
 while (isEmpty())
  try { notEmpty.await();
  } catch (      e) {}
 x = doDequeue();
 notFull.signal();
 lock.unlock();
 return x;
}
```

Critical section

- mutex (must not be interleaved)
- freedom from DL + freedom from starvation

base idea:

```
IRQ.off   // interrupt
critical section   request
IRQ.on
```

```
acquire(monitorLock);
while(! conditionMet) {
  wait(); //*
}
crit.section
signal(conditionVariable);
release(monitorLock);
```

* on this monitor, under conditionVariable

Dekker's Alg for crit. section

```
bool wantP=FALSE, wantQ=FALSE; int turn=1;
PROCESS P                          PROCESS Q
loop
 non crit. section
 wantP = TRUE                      wantQ
 while(wantQ)                      wantP
  wantP = FALSE                    wantQ
  await (turn == 1)                ==2
  wantP = TRUE                     wantQ
 critical section
 turn = 2                          == 1
 wantP = FALSE                     wantQ
```

4.16 - Behind Locks implementation

critical sections (events & precedence)

thread P produces events $b_P, a_P, \dots$ (j^{th} occurrence of event: (es. 3rd iteration in loop, 5th event fired) p_i^j (= p_5^3)

$a \to b$

a occurs before b (precedence, total order relation)

Interval: $(a_0, a_1) \Rightarrow a_0 \to a_1$

$I_A = (a_0, a_1)$, $I_B = (b_0, b_1)$, $I_A \to I_B \Rightarrow a_1 \to b_0$

("I_A precedes I_B", else they are concurrent

Filter lock (extension of Peterson's to multiple processes)

```
int[#threads] level, int[#threads] victim;
lock: for (int i=1; i<n; i++)
        level[me] = i; victim[i] = me;
        while (∃ k ≠ me : level[k] >= i &&
               victim[i] == me) {};
unlock: level[me] = 0
```

Peterson Lock (more concise)

```
P=1, Q=2, bool[] flag=false, int victim=1
PROCESS P
loop
 non-crit section
 flag[P] = true;          // I am interested
 victim = P;              // but you go first
 await (!flag[Q] || victim != P)  // no interest?
 critical section         // I go
 flag[P] = false
```

Bakery (or Post) Algorithm

```
int np=0, ng=0   //example for 2 processes
PROCESS P:
loop
 non-crit. section
 np = ng+1;
 await (ng==0 || np<ng)   // Q doesn't require
   critical section       // access or I got
 np=0                     // an earlier ticket
```

the filter lock is not fair (not FiFo)
so let's use the bakery alg
main idea: each thread takes a number ticket

Hardware support: Read-Modify-Write operations (more powerful than)

- TAS (test and SET)
- CAS (compare and SWAP)
- load linked / store conditional

→ enable mutex implementation in O(1)
→ lock free programming

```
bool TAS(memref s)          // atomic
 if (mem[s] == 0) {
  mem[s] = 1;
  return TRUE;
 } else { return FALSE; }
```

- init(lock) lock = 0;
- Acquire(lock) while(!TAS(lock)); //wait
- Release(lock) lock = 0;

```
int CAS(memref s, int old, new)   // atomic
 oldval = mem[s];
 if (oldval == old) {
  mem[s] = new;
 } return oldval;
```

- init(lock) lock = 0;
- Acquire(lock)
  repeat res = CAS(lock,0,1); until res == 0;
- Release(lock) CAS(lock,1,0); wake up(this);

Spinlock

normal lock that uses loop (busy waiting) to check if lock has been set free. usually inefficient, one can use specific API's. (spinlock don't have notification system)

IMPLEMENTATION OF SPINLOCK
se non avrà senso spinlock in uniprocessore (una sola cpu), se ci sono più cpu allora ha senso

Atomic in Java

```
while(!res.getAndSet(false,true)){
 sleep(this, lock, true)}
```

java.util.concurrent.atomic.AtomicBoolean

AtomicBoolean state = new AtomicBoolean(false);

- get
- compareAndSet(boolean expect, bool update)
- set
- getAndSet(bool newValue)

Backoff

too many threads trying to access same resource → slows down everything
solution: put them to sleep with random duration (increase duration each time res. is not free)

Thread.sleep(time); // fa attendere il thread (meglio di busy loop che spreca CLK)
this is called locks with scheduling (put some to sleep)
→ semaphores, monitors and mutexes use it

4.17 - WITHOUT LOCKS

putting to sleep has also a negative side: higher wake-up latency. possible solution → Hybrid: spinlock at beginning, change in schedule after some time

LOCK-FREE

locks → pessimistic by design. performance: ⊖ overheads ⊖ contented case = slowdown ⊖ bottle neck ⊖ if delayed in crit-s, everyone suffers

lock-freedom: at least 1 algorithm makes progress if others are waiting

wait-freedom: all alg. eventually make progress (no starvation)

non-blocking: block/suspension of thread doesn't affect others (with also blocking or suspension)

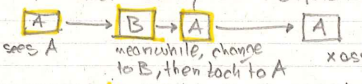
use CAS → read, if no change you are exclusive → write

	non blocking	blocking
all make progress	WAIT-free *1	STARV-free
one makes progress	LOCK-free *2	DL-free

*1 → per thread progress
*2 → system-wide "

ABA problem

activity doesn't recognize that memory location was temporarily been modified by another activity and therefore wrongly assumes it hasn't been changed



(es. stack, looks at only uppermost element!)

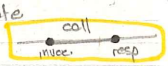
SOLUTIONS:
- DCAS (double CAS)
- pointer tagging (save last bits of address → now $\frac{1}{32}$ chance of fail)
- hazard pointer (before reading, marks it as dangerous)

scheduler is heart of OS

4.18 - CONCURRENCY THEORY

Obj → container for data with set of methods to modify state

Method call → starts with invocation, ends with response

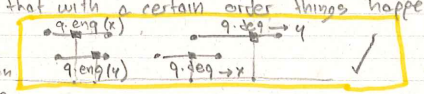


sequential	concurrent
• meaningful state between method calls	• calls overlap, might never be meaningful state
• methods are isolated	• all interactions are possible (must take into account)
• can add new methods without affecting	• "object" CLK
• global CLK	

LINEARIZABILITY

- each method should Appear to take effect instantaneously between invocations and response events

→ object for which is true forall possible executions is called linearizable (simply see if it's possible that with a certain order things happen that way)



Method call: 2 events → • invocation → • response

A q.enq(x) → ack (A: thread, q: obj, enq: method, x: arg, ack: result)
A q.void

History: sequence of invoc & resp (sequential if no method calls interleave)

H = A q.enq(x), A q.void, A q.enq(x) → pending (no resp yet)

A q.enq(x) } match: if invoc & resp match

History is complete if no pending

Projections: Obj-proj H|q (all calls & resp of object q)
Thread-proj H|A (" " thread A)

History:
- well formed (per thread projection is sequential)
- equivalent (each thread projection is equivalent)
- legal (pre/post conditions tell what is legal for obj)
  → if every object is legal, then so is the History
- linearizable (if can be extended to history G by
  - appending x≥0 responses to pending invoc (that happened)
  - discarding y≥0 pending invoc that didn't take effect

such that: G is equivalent to legal seq. history S

Precedence: A precedes B iff A.resp → B.invoc (else overlap)

Composability theorem

H linearizable ⟺ H|obj linearizable (this means modularity)

→ G c → S (G is respects order of G, and also a few limitations/conditions)

Sequential consistency

H is seq. consistent ⟺ same as linearizable, but the "such that..." is not needed (therefore it is weaker)

CONSENSUS

serve a mettere d'accordo su un valore comune usato da più threads (all threads decide same value) → common decision is one of the input

```
public interface Consensus<T> {
  T decide(T value);
}
```

RMW (read modify write)

```
getAndSomeOp() {
 int prior = this.value;
 this.value = someOp(this.value);
 return prior;
}
```

4.19 - from JAVA to C

C is more in touch with basic HW (es memory allocation & management)
Java hides many details for OS writing

Java	C
• OOP • polymorphism	• sequential • limited
• automatic mem. management	• malloc, free
• references	• pointer. (memory address)
• collection of classes	• collection of functions

every data obj has → • type • address • size • lifetime • name • value • visibility

C offers OPENCL, OPENMP, MPI

possible errors while compiling:
- preprocessor (missing include files)
- parser (syntax errors)
- linker (missing libraries)

C preprocessor

↳ takes #statements and substitutes them in the code
(es #define yes 255) → but don't define functions

```
#ifdef name
 code segment 1
#else
 code segment 2
#endif
```

typedef short int smallnumber
↳ (gives names to types)
typedef char boolean;
#define FALSE 0
#define TRUE 1

Arrays: int vec[3];
⚠ doesn't remember how large they are! (sizeof = # bytes) do sizeof A
↳ they are pointers! no checks for bounds sizeof A[0]

int* a = (int*) malloc(len * sizeof(int));

address = a, address = a + i*s (s = size of 1 cell)

struct (like Java, C#)

Pointer* (direct mem address)

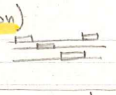
& → provides pointer to data obj (address op)
* → " data obj to a pointer (dereference op)

```
int i=5;
int* iptr = &i;  int j=*iptr; //j=5
```

Void pointer: can be assigned

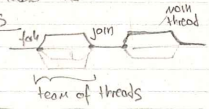
L.20 - Open MP (API for multi-thread application)

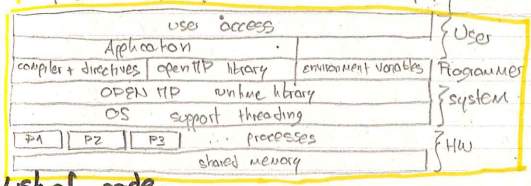
concurrency: multiple tasks are logically active at once
parallelism: " actually "



syntax: compiler directives
#include <omp.h>
pragma omp Construct {clauses}
pragma omp parallel num_threads (4)

threads communicate via shared variables
fork-join:





false sharing = cache inconsistency when saving

List of code

```
#include <omp.h>
omp_set_num_threads(8); // how many
# pragma omp parallel {   // what's inside will be executed parallel
  int id = omp_get_thread_num(); // inside the block gets unique thread id
}
int omp_get_num_threads(); // # in current team
double omp_get_wtime();    // wall clk in seconds

# pragma omp barrier      //simple barrier
# pragma omp critical     //can only be executed by 1 at a time
    (res += consume B;)
# pragma omp atomic
# pragma omp for
    for (i=0,...,-)
```

fork / join

locks

```
omp_init_lock();
omp_set_lock();
omp_unset_lock();
omp_destroy_lock();
```

sections

```
# pragma omp master
  //code executed by master thread (id=0)
# pragma omp single
  //executed by 1 thread
# pragma omp section
# pragma omp task
```

Static vs dynamic schedule

static: fixed size chunks
dynamic: variable "

Reduction

storage attributes: → all 3 mean local variable (otherwise it's shared by all)
- shared
- private // ⚠ private var cannot initialized, cannot be used outside scope
- firstprivate (in) } propagated in/out of scope
- last private (out) }
- default

* provides private copy of global variable to each thread

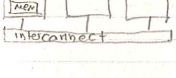
```
# pragma omp thread private (variables) //*
# pragma omp parallel shared (A,B,C) private (id) {...}
                declaration of variables
# pragma omp parallel for reduction (+: ave){
    for (int i=0; i<MAX; i++)
      ave += A[i];
    ave /= MAX;   //average.
```

↔ local copy is made and initialized according to "op"
(op: variable)
+/- → 0
* → 1
min → float.max
max → float.min

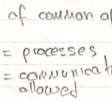
task: independent unit of work (
- code to execute
- data environment
- control variables)

21. OPEN MP & MPI

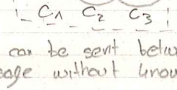
MPI: message passing interface (a mechanism to send/receive messages)
distributed memory system



SPMD (single program multiple data)



MPI communicators: defines common domain of common op.



MPI_COMM_WORLD p = processes, c = communications allowed

since many diff messages can be sent between two guys, tag is used to differentiate
receiver can get message without knowing sender, tag, amount of data
reception status

Synchronous message passing: sender waits until whole message can be accepted by receiving process, before sending
receiver waits until he gets message (used for data transfer & sync)
Asynchronous mess passing: send/receive do not wait
this requires local storage for messages

about communication S-R

Blocking: return after local actions are complete (only then) } about local handling of sending/receiving
Non b.: return immediately

Interface

```
#include <mpi.h> // use the header
MPI_Init (&argc, &argv);
MPI_Comm_size (comm, &size);
MPI_Comm_rank (comm, &rank); //id, number

int MPI_Send (void* msg_buf_p,   //pointer to data to sent
              int msg_size,      //# items to be sent
              MPI_Datatype msg_type, //type
              int dest,          //destination
              int tag,           //data identification tag
              MPI_Comm communicator)

int MPI_Recv (void* buf_p,
              int buf_size,
              "
              int source,        //source process id
              "
              MPI_Status* status_p); //status
```

they have to match!

eg. MPI_Send (&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Send is - blocking (sync/async depends on implementation)
There also are variations for block/non, sync/an, buffered/non, ... (MPI_Bsend, MPI_Ssend, MPI_Rsend...)
most implementations only allows process 0 (main) to access important stuff (like stdins to read data)

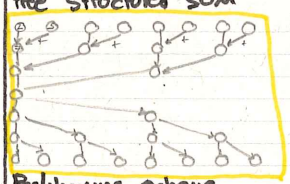
Parallelization steps

1) partition solution into tasks
2) identify comm. channels between tasks
3) aggregate tasks into composite
4) map it to cores

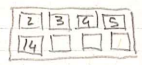
Collective Communication

until now, we have used point 2 point comm.
you can also send mess between groups of processes eg. MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Scatter, MPI_Gather, ...

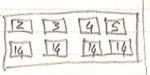
tree structured sum

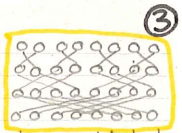


only this part is needed for sum (computation)

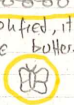


broadcast (no computation, just copy)





when simplified, it becomes the butterfly process



Partitioning scheme

- Block (MPI) - blocks of consecutive components to each process

0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

- Cyclic - assign in round robin fashion

0	0	3	6	9
1	1	4	7	10
2	2	5	8	11

- block/cyclic

process				
0	0	1	6	7
1	2	3	8	9
2	4	5	10	11

MPI_Reduce (

```
void* input_data_p,
void* output_data_p,
int count,
MPI_Datatype datatype,
MPI_Op operator,
MPI_Comm comm );
```

e.g. MPI_Reduce (&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

possible operators: • MPI_MAX • MPI_SUM ...
• MPI_MIN • MPI_PROD

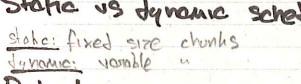
⚠ all processes in communicator must call same funct
⚠ args passed must be compatible

other methods

MPI_Allreduce — copia il risultato in tutti
MPI_Broadcast — data belonging to one of the processes is sent to all processes in communicator
MPI_Scatter — reads in an entire vector, then sends only the required parts to the other processes
MPI_Gather — collect all components of vector into destination process, then this can process all components
MPI_Allgather — gather + broadcast (same everything in everyone)

you can use this to create a system for matrix multiplication

L.22 - Intro to massively par. prog (GPGPU)

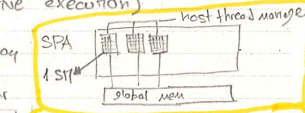


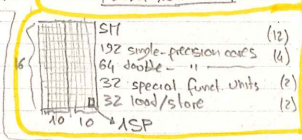
4 cores — 1536 (or 33x39) cores
500 vs 4500 gflops (Gigafloat op/s)
40 vs 170 GB/s bandwidth

we can no longer increase CLK frequency • consumption • heat • too long path
More cores on chip requires smaller, simpler cores
GPU power makes them interesting for general purpose computation. In the past, very difficult to use them, due to • rapidly evolving • secret • inherently parallel

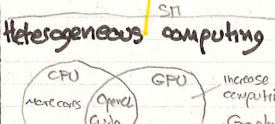
CPU: focuses on low latency (out of order & speculative execution)
GPU: focuses on throughput

terminology:
SPA (device): streaming processor array
SP: streaming processor
SM: streaming multiprocessor





Heterogeneous computing



increase general purpose data-parallel computing
Graphics/shading

Cuda: (compute unified device architecture) is a general purpose programming model

OpenCL (open computing language)
is an open standard for heterogeneous programming
targets wider range than cuda
(less peak performance than specific cuda)
• cross platform parallel computing API's, C like language
• language specification for kernels (each kernel is executed by compute device)

es. GPU: fixed graphics pipeline

Granularity:
Coarse: process complex on a large block of data before communication
Fine: individual process quite small, more communication

L.23 - GPGPU

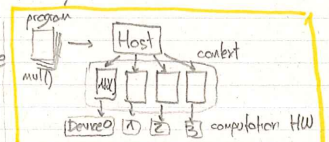
Host program is written in high-level language (C/C++)
kernels written in open CL

Vocabulary:

platform (vendor's implementation of openCL) manages collection of devices
device piece of HW accessible via vendor's platform (with work to be done)
context collection of devices (same platform) owned at working together + command queue
kernel smallest unit of execution for device (work item) - C function + part of program
program collection of kernels (stored on file)
work item instance of kernels at runtime (runs in work group). has unique ID
work group collection of work items. has unique work ID

OpenCL & cards analogy

Structure of openCL program



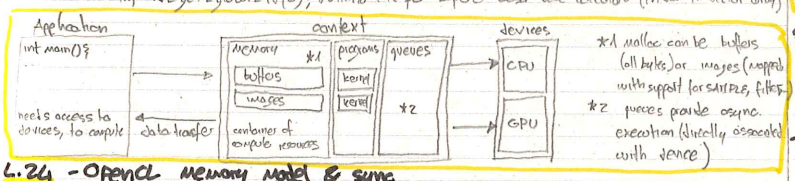
1) set info about platform & devices available
2) collect devices in context
3) create openCL command queue in context
4) create new buffer on device
5) transfer data from host to device mem. buffer
6) create kernel
7) build/compile kernel
8) create openCL kernel obj
9) set kernel args
10) execute kernel
11) read kernel mem and copy to host mem.

Data parallelism: define N per computation domain (it defines ter
CPU coordina, GPU calcola # of work ite
Δ problems that map well (independent; scaling (color/texture), lin. algeb
vs not so well (sequential, dependent)
Array of work items (execute kernels) SIMD
→ work items within work group cooperate (shared mem, atomic

API

```
cl_platform_id* platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id)*length); //get platform IDs ①
res = clGetPlatformIDs(len, platforms, &nPlatforms);
res = clGetPlatformInfo(platforms[i], CL_PLATFORM_EXTENSION, 0, null, &ext_size); //get platform info
res = clGetDeviceIDs(...);        clGetDeviceInfo(...)        //get device IDs info
cl_context context = clCreateContext(properties, usedDevices, deviceIDs, null, null, &res); //create context ②
cl_command_queue commands = clCreateCommandQueue(...);        //create comm. queue ③
cl_mem inputBuffer = clCreateBuffer(...)                      // create buffer ④ ⑤
cl_program program = clCreateProgramWithSource(...); res=clBuildProgram(...); //create & compile program ⑥ ⑦
cl_kernel kernel = clCreateKernel(...);                       //return kernel to execute ⑧
res |= clSetKernelArg(...);                                   // set kernel args ⑨
res = clGetKernelWorkGroupInfo(...);                          //schedule kernel for execution ⑩
clFinish(commands); clEnqueueReadBuffer(...);                 // read result ⑪
```

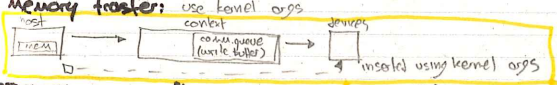
Domain config: 1D (eg. elem in array) 2D (eg. image) 3D (ex. voxels)
Host: create & manage device context - associate work } compiled at runtime (by host)
• create queues • memory alloc • kernel launch
Data types: • scalar (normal) • vector (es. bool4)
there are object creation & management methods extensions (-kernel vaad...)
int id = get_global_id(0); //ottiene l'id per sapere cosa deve calcolare (index in vector array)



L.24 - OpenCL memory model & sync

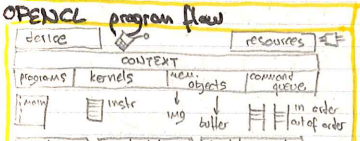
code can be compiled for different devices (CPU, GPU, ..)

Memory transfer: use kernel args

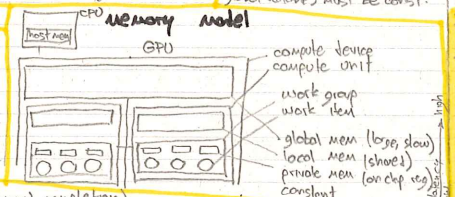


pointer args can be everything but private
global variables must be const.

OPENCL program flow



CPU memory model



Events
• host notification (eg. command completion)
• command sync • profiling
↳ can be associated to callback function (called when event happens)

usually, the elements in context command queue are sent to devices in out of order. You can't control it. but you can constraint a command to wait for another to be executed.
you can obtain profiling info ⚠ host mem can be mapped to device mem, to save all data transf (very efficient, but not always the best choice)

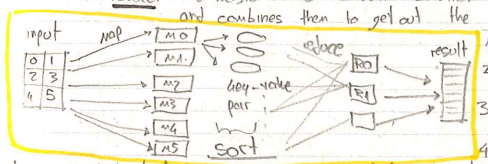
KERNEL LEVEL SYNC MECHANISM
memory model uses relaxed consistency • if one data is visible to one work item, it's not guaranteed to be seen by all in work group
• you need sync for that

GLOBAL & LOCAL: # global work items limited by size_t
work items per group limited by max work group size
items have 2 ID's: global & local

workgroupsize: cl GetKernelWorkGroupInfo(...);
memory organisation: loc. mem organized into mem. banks (usually by 16/32)

L.25 - OpenCL case studies

Map & reduce is a strategy for implementing parallel algorithms
• map: a master worker takes problem input, divides into subproblems, divides to threads
• reduce: the master worker collects solution of workers and combines them to get out the final solution



1) each work item in work group performs mapping
2) barrier prevents further execution until done
3) local reduction: each localID == 0 work item (1st in group) performs reduction
4) global barrier
5) work item with globalID==0 performs final reduction

⚠ OpenCL doesn't feature dynamic malloc! You cannot know in advance # of value-pairs
⚠ necessario sync work group, anche in mapping

Sorting heap & merge sort → $O(n \log n)$ worst
quick sort → $O(n \log n)$ average, $O(n^2)$ worst

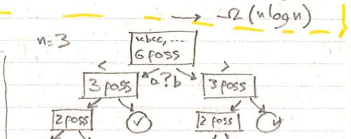
impossible to go lower!
permutations: $n!$ these are all comparison sorting. At best, we eliminate $\frac{1}{2}$ of possibilities while searching for something.

binary: each question → 2 outcomes (assuming no duplicates/redundant questions)
every answer is a leaf → the tree has to be big enough to contain all
⇒ no comparison alg can be better than this running time

proof: height of bin. tree with $n!$ leaves $\in \Omega(n \log n)$
lower bound on height: bin tree with L leaves has at least height $\log_2 L$

$$h \geq \log_2(n!) = \log_2(n(n-1)(n-2)\dots) = \log_2 n + \log_2(n-1) + \dots + \log_2 1$$
$$\geq \underbrace{\log_2 n + \log_2(n-1) + \dots + \log_2\left(\frac{n}{2}\right)}_{\frac{n}{2} \text{ terms}} \geq \left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) =$$
$$= \frac{n}{2}(\log_2 n - \log_2 2) = \frac{1}{2}(n \log n - n)$$
$$\rightarrow \Omega(n \log n)$$

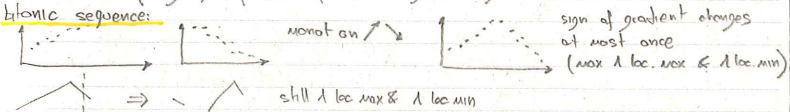
node tree: each node = remaining possibility



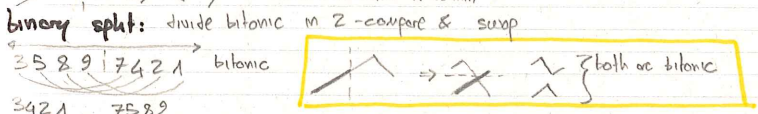
Horrible alg: $\Omega(n^2)$ → bogo, stooge
simple alg: $O(n^2)$ → insertion, selection, bubble
fancier alg: $O(n \log n)$ → heap, quick, merge
specialized: $O(n)$ → radix, bucket

Bitonic: $O(n \log^2 n)$ - sequential
$O(\log^2 n)$ - parallel (worst = average = best)

bitonic sequence:



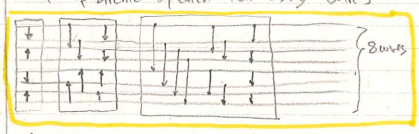
binary split: divide bitonic in 2-compare & swap



since it was bitonic, every element in first partition is smaller than element in 2nd ⇒ recursively "bitonic splitting" will sort

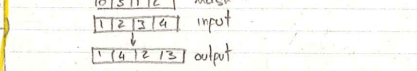
forming bitonic from scratch:
1) pairs of adjacent numbers already bitonic → merge in longer sequence
2) compare first 2 elem (half?). If 1st > 2nd, swap. ensure ascending order
3) search " < swap. descending

example of bitonic implementation using wires



input → output
the arrows are comparators. the 2 numbers are compared to ensure small → big. else swap.

Shuffle: take elements in order from mask



END

Good vocabulary: threads interleave inconsistent state

volatile: becomes visible to readers once writer has finished (otherwise they would see some non-updated value)

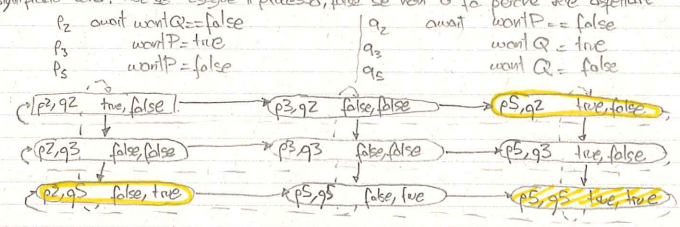
⇒ see wiki "monitor sync"

Check for possible problems ⚠
Deadlock / Starvation
Inconsistency (Stale)
Barrier
Atomicity
Race Condition
Visibility
Global / Local
Off by 1 / Fencepost
order of execution
Interleaving
granularity (coarse vs fine)
ABA problem (CAS)

CPU	GPU
low latency cache access	data parallel throughput
high throughput	low latency
complex control logic ↑ (more space dedicated to)	
lot of cache space	ALU: lot of dedicated space

State space diagram
for un diagramma a stati bisogna tutte le possibilità (quasi come una matrice con frecce)
if wantP=T, wantQ=T must fail → no mutual exclusion. If this happens inside "while", DL
significato della: true se sto eseguendo il processo, false se non lo fa perché deve aspettare

p_2	want wantQ==false	q_2	want wantP==false
p_3	wantP=true	q_3	wantQ=true
p_5	wantP=false	q_5	wantQ=false



Non blocking counter

```
public class CasCounter {
  private AtomicInteger value;
  public int getVal(){
    return value.get();}
  public int inc(){   // increment and return new value
    int v;
    do { v = value.get();
    } while (! value.compareAndSet(v, v+1));
    return v+1; }}
```

Mechanism: (a) read current value v. (b) modify value v'. (c) try to set with CAS (d) return if success, restart at (a) otherwise. ⚠ positive result of CAS of (c) suggests that no other thread has written between (a),(c).
↳ not always true, ABA problem (but it's the way to control exclusive access in lock-free prog.

Threads (x thread)

```
public class TaskThread {
  public static void main (String[] args){
    TaskClass task = new Task Class(...);
    // or Runnable "
    Thread thread = new Thread (task);
    thread.start(); }}
public TaskClass implements Runnable {
  ... public TaskClass(...) {...}
    public void run() {
    // implement run if runnable and tells how to
    run custom thread
    ...}...}

class ... { main ... {
  ExecutorService ex = Executor.newCachedThread Pool();
  //create and launch 100 thread
  for(int i=0; i<100; i++){
    executor.execute(...); }
  executors.shutdown();
  //wait all tasks are finished
  while(! executor.IsTerminated()) {}}
other class is same as thread xthread
(override run)
```

Non blocking stack

```
public static class Node {
  public final long item;
  public Node next;
  public Node (long item){
    this.item = item; }}
public class ConcurrentStack {
  AtomicReference<Node> top = new A...<N..>();
  public void push (long item){...}
  public long pop(){...}}
public long foo() {
  Node head, next;
  do { head = top.get();
  if (head == null) return null;
    next = head.next;
  } while(! top.compareAndSet(head, next));
  return head.item; }
public void push (long item){
  Node head;
  Node newi = new Node(item);
  do { head = top.get();
    newi.next = head;
  } while(! top.compareAndSet(head, newi)); }
```

Fork join Pool: Parallel Max

```
public class ParallelMax {...
  public static int max (int[] list){
    RecursiveTask<Integer> task = new MaxTask(list,0,
    ForkJoinPool pool = new ForkJoinPool();
    return pool.invoke(task); }
private static class MaxTask extends RecursiveTask
public MaxTask(int[] list, int low, int high) {...}
  @override
  public Integer compute() {
  if...   max = ...
  return Integer(max);
  else
    RecursiveTask<Integer> left = new MaxTask<Int>,
                          right = "
    left.fork();   right.fork();
return new Integer(Math.max(left.join(), intValue(),
                   right.join(), intValue()) );}}}
```

OpenCL

```
-- kernel void
matrixMul(--global float*C, "A, " B,
          int wA, int wB){
int tx = get_local_id(0); // 2D thread
int ty =      "      (1); // 1D  "
float value=0;
for(int k=0; k< wA, ++k){
  float elemA = A[ty*wA + k];
  float elemB = B[k*wB + tx];
  value += elemA*elemB;}
C[ty*wA + tx] = value; }
```